Static Analysis of the Mars Exploration Rover Flight Software

Guillaume Brat
Kestrel Technology
brat@email.arc.nasa.gov

Roger Klemm
California Institute of Technology
roger.klemm@jpl.nasa.gov

Abstract

The demise of the Mars Orbiter and Mars Polar Lander missions has highlighted the criticality of software reliability for Mars missions. In both cases, problems manifested themselves at the software level, even if the causes are to be found somewhere else (e.g., design process). Therefore, it is fair to assume that many problems could be caught during software verification provided that one uses the right tools and looks for the right types of errors. In this paper, we describe a study in which we apply this theory to the flight software of a current mission, i.e., the Mars Exploration Rover mission (MER). The study consists of applying a static analysis tool to the MER code to identify runtime errors, such as un-initialized variables, out-of-bound array accesses, and arithmetic overflows and underflows. The goal is both to demonstrate the usefulness of formal methods in a real software development context, and more importantly, to participate in the verification of the code that will fly during this mission. The work was conducted by a tool expert and a code expert. All identified problems were passed on to the appropriate developers. This paper describes the setup of the study, the findings, and proposals for integrating such a tool in a software development process. It also includes illustrative examples of the problems found by the analysis.

1. Introduction

Designing reliable software systems is a critical need for NASA. The result of a software fault ranges from losing a day of science data (as in the Mars Path Finder, MPF) to the loss of an entire mission (e.g., Mars Polar Lander, MPL). Losing an entire mission results in wasting precious time in scientific experiments; e.g., one has to wait two years to find another suitable launch window for Mars, but it takes at least three years to prepare hardware and software for a new mission. The financial impact is also significant; e.g., a rover mission to Mars costs a minimum of $250 million dollars. It is easy to see why NASA has a vested interest in increasing the reliability of its critical software systems.

In general, the software development process depends greatly on the current flight software manager. However, the verification and validation process (V&V) is more or less the same across missions. It consists of some unit testing performed by the developers themselves, and then a series of system integration tests using different levels of simulation. For some missions, the actual hardware was preferred to even a high-fidelity testbed. We are not aware of the use of any formal methods in the V&V process. In some sense, one can say that the validation aspect takes precedence over the verification aspect. Our overall goal is to try to convince mission developers that significant reliability gains can be obtained by inserting some formal methods in the V&V process.

For such a demonstration to be effective, it needs to be conducted on real code with real errors. Then, one has a chance to convince developers that automated tools based on formal methods can identify cases that elude any tester. For this particular experiment, we chose to concentrate on one technique (i.e., static analysis, and more specifically, static analysis based on abstract interpretation) and one specific class of errors (runtime errors such as out-of-bound array accesses, un-initialized variables, arithmetic overflows and underflows, and so on). We applied this technique to the flight software system of the Mars Exploration Rover mission. Note that the goal is not to realize a statistically meaningful study, but to demonstrate the usefulness of current static analysis.

2. Static Analysis

The goal of static analysis is to assess code properties without executing the code. Several techniques can be used to perform static analysis, such as theorem proving, data flow analysis [1], constraint solving [2], and abstract interpretation [3,4]. For this experiment, we use a tool, called PolySpace C-Verifier [5], which is based on abstract interpretation.

2.1. Overview

The theory of Abstract Interpretation pioneered by Patrick and Radhia Cousot in the mid 70's provides

algorithms for building program analyzers which can detect all runtime errors by exploring the text of the program [3,4]. The program is not executed and no test case is needed. A program analyzer based on Abstract Interpretation is a kind of theorem prover that infers properties about the execution of the program from its text (the source code) and a formal specification of the semantics of the language (which is built in the analyzer). The fundamental result of Abstract Interpretation is that program analyzers obtained by following the formal framework defined by Patrick and Radhia Cousot are guaranteed to cover all possible execution paths.

Runtime errors are errors that cause exceptions at runtime. Typically, in C, either they result in creating core files or they cause data corruption that may cause crashes. In this study we mostly looked for the following runtime errors:

- Access to un-initialized variables (NIV)
- Access to un-initialized pointers (NIP)
- Out-of-bound array access (OBA)
- Arithmetic underflow/overflow (OVF)
- Invalid arithmetic operations (e.g., dividing by zero or taking the square root of a negative number) (IAO)
- Non-terminating loops (NTL)
- Non-terminating calls (NTC)

The price to pay for exhaustive coverage is incompleteness: the analyzer can raise false alarms on some operations that are actually safe. However, if the analyzer deems an operation safe, then this property holds for all possible execution paths. The program analyzer can also detect certain runtime errors which occur every time the execution reaches some point in the program. Therefore, there are basically two complementary uses of a program analyzer:

- as a debugger that detects runtime errors statically without executing the program,
- as a preprocessor that reduces the number of potentially dangerous operations that have to be checked by a traditional validation process (code reviewing, test writing, and so on).

For the second use the tool should achieve a good selectivity - the percentage of operations which are proven to be safe by the program analyzer. Indeed, if 80% of all operations in the program are marked as potentially dangerous by the analyzer, there are no benefits to using such techniques.

2.2. PolySpace C-Verifier

PolySpace C-Verifier is the first tool implementing Abstract Interpretation techniques that is able to scale up to software systems of industrial size. This tool takes an ISO-compliant piece of C code and performs static analysis using sophisticated Abstract Interpretation algorithms. The result is the program in which all potentially dangerous operations have been assigned a color:

- **Green** the operation is safe, no runtime error can occur at this point
- **Red** a runtime error occurs whenever the operation is executed
- **Black** the operation is unreachable (dead code)
- **Orange** the operation is potentially dangerous (runtime error or false alarm).

PolySpace C-Verifier achieves a fairly good selectivity, since in practice no more than 20% of operations are marked as orange. In this paper we focus mostly on the operations that are marked as red by the analyzer. It means that we use the analyzer as an "abstract debugger" which points out errors without running the program.

3. Code Preparation

Ideally, a verification tool should accept code as it is written by the developer. However, it is not necessarily the case in practice. The code always has to be massaged a little before it goes through. We first give an overview of the MER code, and then we describe the modifications we made to accommodate the analysis.

3.1. MER Code Structure

In this section, we give a high-level view of the MER code. MPF was a successful mission, and therefore, its code has been used as a base for developing the code for other missions such as Deep Space One, DS1, and MER. The software is built on top of the VxWorks operating system. All MPF-derived code we analyzed is organized as follows.

The code is multi-threaded. All threads are created during an initialization phase, but they are activated only when needed. For example, there is no need to activate the EDL (Entry, Descent, and Landing) thread before the end of the cruise phase. Threads communicate through message passing using mechanisms given by the VxWorks OS. Each request message provides the name of a callback routine to return results. There are more than 100 tasks in MER.

In general, the software keeps two logs during computations. One stores scientific data, while the other keeps track of the safe operation of the system in terms of event sequences. The logging rate depends on the storage capacity, the criticality of the current computations, and the bandwidth available to send these data to ground control.

State, and critical, information is stored in fairly shallow data structures which are allocated during the initialization phase. Typically, these data structures are arrays (they represent the matrices that are used for controlling the spacecraft), or records whose fields may be arrays. These data structure have a nested depth of two or three, which is why we refer to them as fairly shallow.

3.2. Code Modifications

The first modification is due to detecting errors. These errors most often come from the compilation phase. Indeed, PolySpace Verifier checks for strict ISO standards, when traditional compilers are more permissive. Sometimes (and hopefully) errors are discovered by the analysis, and they need to be fixed before the analysis can be run again. In both cases, these code modifications are necessary.

The second type of modifications comes from scalability problems. Big software systems have to be divided into pieces ranging from 20 KLOCS to 40 KLOCS. Sometimes this decomposition is natural because of the modularity of the code, sometimes it is not obvious. Stubs for the environment may have to be written.

The third type of modifications is due to the tool limitations. For example, the tool assumes a true concurrency models in which all threads are started as soon as the main ends; this resulted in minimal modifications. Basically, we wrote a main program that performed the initialization needed by the modules we analyzed. The Mars code uses quite a bit of function pointers, and the tool could not always resolve those. In these cases, the code had to be carefully modified to ensure the validity of the results.

4. Analysis Results

In this section, we describe the results we obtain during the experiments. All analyses were conducted using the PolySpace Verifier on a PC (running Linux) using a 2 MHz CPU and 2 GB of memory. In a subsequent section, we give approximate performance figures. First, we describe the types of errors we encountered. Obviously we cannot mention all the errors we found and their exact description. Finally we give performance figures.

4.1. Results

The MER code suffered essentially from NIV problems. We mention only the following one because it will lead to an interesting discussion in the next section.

```
void getData (T* p) {
   ...
   if (flag == TRUE) {
       ...
       p->data = ...;
       p->status = 1;
       ...
   }
   else {
       sendEvrMsg("data unavailable");
   }
}
```

It is obvious that the getData routine can exit without setting the contents of the structure passed as an argument.
Therefore, the tool flagged numerous uses of variables passed to getData when their data field was accessed. Note that the status field in type T is there to tell if the data is available or not. So, every access to this structure should be preceded by a test of the status field. It was not the case in the version we analyzed. However, even if it were the case, we would have had an error on the test of the status field unless it had been set to a default value at initialization.

Our final error example was caught very quickly by the tool. In fact, it was caught during the compilation phase without requiring running any analysis algorithm. In this case the tool noticed that the address of a local variable was returned.

```
int foo(...) {
    int var;
    ...
    var = ...;
    ...
    return &var;-----------
}
```

4.2. Performance

There are usually two problems with using static analysis on real programs. First, static analysis tools have trouble scaling to large software systems (100 KLOCS and up). Second, static analysis is usually conservative (all execution paths are covered), and this results in the generation of many false alarms. We now discuss both aspects based on our experience.

PolySpace Verifier was originally conceived for Ada programs. It seems that it scales well for such programs (over 1 Million LOCS). Unfortunately, the code we analyzed was written in C, and was too large (more than

650 BLOCS) for the tool. We experienced a limitation of about 40 KLOCS given the type of algorithmic and structure complexity present in the MER code. To get around this scalability problem, we divided the code into modules ranging from 20 KLOCS to 40 KLOCS. We intentionally kept complicated modules close to 20 KLOCS so that we could obtain decent precision. This means that the rest of the program was stubbed. In general, we chose to stub utility modules that implement the ``plumbing'' of the system (i.e., communication layer, file I/O, and so on). We concentrated our analyses on the critical modules and on modules that were either significantly different from the MPF code or new. Our reasoning here is that utility modules are often inherited without modifications from one mission to the next. Critical components such as the attitude control module are usually one of a kind.

We measure precision (also called selectivity by PolySpace) of the analysis in terms of the distribution of the safety checks performed by the tools. Thus, our precision measure is given by the following formula:

$$\text{precision} = (G + R + U) / (G + R + U + O)$$

where G is the number of checks deemed safe (Green), R the number of unsafe (Red) checks, U the number of unreachable checks, and O the number of potentially unsafe (Orange) checks. Throughout our experiments, we obtain a precision ranging from 80% to 90%. This means that between 10% and 20% of the checks have to be classified using other methods (manual inspection, testing, or other technique). This was quite satisfactory, but it appears to be still a deterrent for developers at NASA. This is understandable when one considers that even a small MER module of only 30 KLOCS can generate about 5000 checks; hence, more than 500 checks still have to be verified through other means. Such high numbers usually scare developers away from using this type of tools. In the next section, we suggest other ways of using the tool. Note that the current research on static analysis based on abstract interpretation shows that the next generation of tools might achieve near 100% precision. For example, Cousot *et al*. analyzed 10 KLOCS of Airbus code with 100% precision [6].

The time needed to run an analysis is also another important aspect that developers worry about: the shorter the analysis time, the more likely the tool will be used. In our study, analysis time was often a problem (especially with early versions of the tool). PolySpace Verifier offered essentially two modes of analysis. The first one is quick (about 30 minutes of analysis time), and it does not offer much precision. It is intended to catch obvious errors before committing to analyses with higher degrees of precision. As the previous section showed, this phase still catches errors despite its relative simplicity. The error about returning a local address was caught during a quick analysis. So, this analysis has its place to weed obvious errors. In general, each subsequent analysis pass took about one hour for the size of modules we analyzed. Note that, each pass builds on top of the results of the previous pass. Since in general after three passes we saw no precision improvement anymore, one can say that a precise analysis requires on average three hours. Therefore, the tool was used mostly in a batch mode rather than an interactive mode.

5. Lessons Learned

In this section, we discuss how to make the most efficient use of static analysis. We emphasize two aspects. First, we give hints about what coding practice will facilitate static analysis, and therefore, will lead to verifiable code. Second, we discuss the place of static analysis in the development process.

5.1. Coding for Static Analysis

Developers need to adapt their coding habits so that their programs can be analyzed more easily, and more precisely, using static analysis. In this section, we point to some examples of coding practice that should be avoided.

One of the trickiest coding practice we had to analyze is the use of function pointers. In the Mars code, this practice appears in two places when a command is issued. The code is organized in such a way that requests for services (in other words control commands) are issued using messages. These messages are handled by a module that consists of a loop that performs some action depending on the type of the message. In order to keep messages generic, pointers to the routine that will perform the service are passed in the message. This pointer corresponds to an entry in a table of service routines. However, since the alias algorithm does not distinguish among elements of an array, the analyzer considers that all service routines can be called. This results in significant approximations. Moreover, the acknowledgment that a service has been performed is implemented through the use of callback routine handles given in the message requesting the service. Once again, this is a source of approximations.

Another dangerous practice consists of re-using blocks of allocated memory and overlaying different types of data structures over these blocks. This is a common practice in embedded systems. Memory allocation is considered a source of unpredictability. Therefore, it is only done during an initialization phase. Initially allocated blocks are then re-used by the program as needed. For example, in MER, messages are implemented using this scheme. Problems arise when blocks are used

sometimes as blocks of integers or sometimes as blocks of floats. Worse, sometimes block of integers hide addresses (that have been cast as integers). This coding practice may be convenient for embedded systems, but it confuses static analyzers which cannot rely on a clean, consistent type system.

Finally, we would like to come back to one of the errors we found. Recall that a data structure had two fields, one holding data values and the other giving a status on the availability, or validity, of the data. The error was that the data field was accessed without being set. The proper use of this coding construct is that the status flag should always been set by the routines that may write in the data field, and the use of the data field in other routines should always be protected by a test on the flag as follows:

```
...
if (p->status == VALID) {
   data = p->data;
   ...
}
...
```

From a static analysis point of view, it is difficult to track the value of the status flag. Therefore, the use of the data field will always be flagged as an orange even though the code is absolutely correct. This example is typical of cases where a static analyzer needs to be able to recognize safe coding patterns. Another solution would consist of providing annotations. However, we do not recommend such solutions since potential users are generally averse to writing annotations, especially in development environments with stringent time constraints.

5.2. Using Static Analysis

The final point we wish to address is when static analysis should be used. Does it belong in development phases or unit testing or system integration? Where is it the most effective? To answer this question, we will rely on the traditional V diagram representing software lifecycle as shown in Figure 1.

We recommend the use of static analysis from the Software Detailed Design phase to the Software Integration phase if possible. Current static analysis tools can be applied without any problem from Software Coding to Software Unit testing. Current scalability problems prevent their full use in the Software Integration phase.

- **Software Coding:** Static analyzers can be used as sophisticated compilers because they perform stricter checks with respect to the ISO standard and they use advanced alias algorithms. They can also be used as abstract debuggers in this phase.
- **Software Unit Testing:** This is the phase in which current analyzers will perform at their best. Units can be verified for a wide range of inputs for a low cost.
- **Software Integration:** This is currently beyond the reach of commercial static analyzers. They cannot scale to full size systems and deliver enough precision. However, they can still be used as abstract debuggers in this phase.

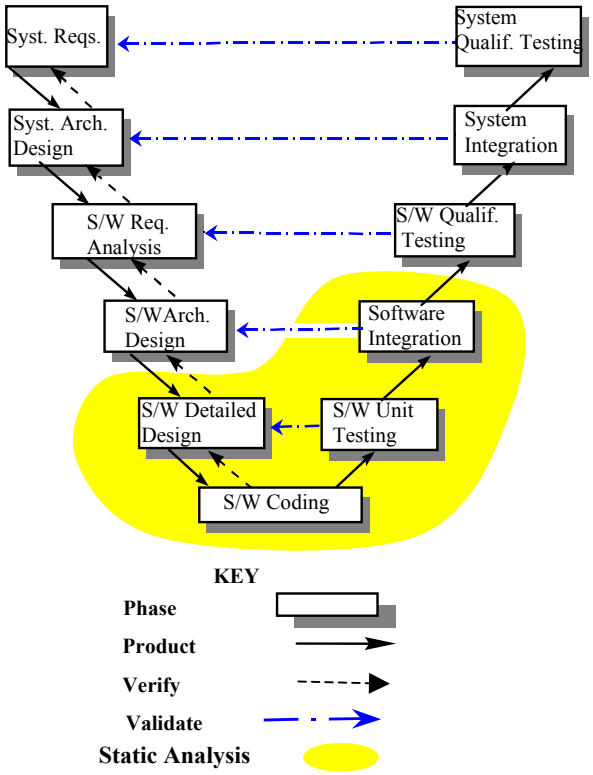

Figure 1. Place of Static Analysis in S/W Lifecycle.

The use of static analyzers as debuggers is interesting; it provides a new way to approach oranges (false alarms). Basically it works as follows. All red errors (unsafe checks) are of course corrected. Then, only some oranges are analyzed. Based on previous experiences, one might decide to look for certain types of oranges (e.g., NIV errors). Another approach is to look for isolated oranges. It is rare that analysis approximations lead to isolated oranges. It usually leads to a cluster of orange checks. So, an isolated orange might be the sign of a deeper problem. Whatever selection criterion is used, the main point is that not all oranges are being checked manually. Only those

that have a serious potential for being true errors (which is determined empirically) are studied.

6. Conclusions

In this paper we have presented the results of a study in which a static analysis tool was applied to real NASA mission code (i.e., modules of the flight software for Mars Exploration Rover). The main results are as follows.

- Current commercial analyzers do not scale to software systems of more than 40 KLOCS.
- They are still useful to find errors since we found errors in real mission code.
- The rate of false alarms has been found to be between 10% and 20%.
- Some coding constructs should be avoided in order to make a more efficient use of static analyzers.
- Aliasing is definitely a bottleneck in terms of precision and scalability.
- Static analyzers can be extremely useful when used as abstract debuggers rather than certification tools.

The goal of this study was to identify the current strengths and weaknesses of static analysis when it is applied to real critical software (as it is built at NASA). Based on this experiment, it is our belief that current weaknesses (namely scalability and precision) can be addressed by building specialized static analyzers that are dedicated to specific software families. To prove our point, we are currently ongoing the design of such a specialized analyzer for the NASA software systems following the Mars Path Finder legacy.

7. References

[1] W. Landi, "Interprocedural Aliasing in the Presence of Pointers", *Ph.D. thesis*, Rutgers University, 1992.

[2] A. Aiken and M. Fähndrich, "Program Analysis using Mixed Term and Set Constraints". In *Proceedings of the 4th International Static Analysis Symposium* (SAS'97), 1997.

[3] P. Cousot and R. Cousot, "Static Determination of Dynamic Properties of Programs". In *Proceedings of 2nd International Symposium on Programming*, pages 106-130, 1976.

[4] P. Cousot and R. Cousot, "Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In *Proceedings of 4th Symposium on Principles of Programming Languages*, pages 238-353, 1977.

[5] PolySpace: http://www.polyspace.com/.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "Design and Implementation of a Special-Purpose Static Program Analyzer for Safety Critical Real-Time Embedded Software". In *the Essence of Computation: Complexity, Analysis, Transformation*, LNCS 2566, pages 85-108, 2003.